MCU to Intel architecture conversion

Lee Eng Kean, Intel Corporation - May 31, 2010

Background

About 8.3 billion microprocessors were produced worldwide in 2009. However, microprocessors for information processing such as personal computers (PC) is about 0.15 billion (which is only 2% of the total sold). The remaining 98% is used for embedded systems.

A real-time system is required to complete processing in a timely and reliable manner. Its output results must be produced in response to input conditions within a deterministic time limit (often in a range of micro seconds). If the results are not produced within that time limit, unpredictable consequences will occur, potentially damaging the system or the people relying on that system.

Programmable Logic Controller (PLC) is one of the real-time control systems widely used for electro-mechanical process automations such as machinery control in factory assembly lines. Unlike consumer applications, electro-mechanical automations require real-time processing capability. A microcontroller (MCU) is commonly chosen for PLC implementation due to its simple architecture which ensures real-time interrupt response.

Migration Considerations

Architecture migration takes multiple areas into consideration, including hardware architectural differences, operating systems, system firmware, and migration/development tools. Another architecture aspect to be considered when migrating from MCU to Intel® Atom™ microarchitecture is moving from a uni-processor serial code to a software system that supports Intel® Hyper-Threading Technology. This paper mainly focuses on hardware interrupt architectural analysis which is an essential consideration for real-time industrial control applications.

Microprocessor vs. Microcontroller Architecture Differences

A microprocessor (MPU), such as the Intel® Atom™ processor or Intel® Core™ i7 processor, contains no memory, graphics, or peripherals on the processor itself. For this reason, they are usually referred to as general-purpose microprocessors.

Memory, graphics, and peripherals (external to the processor) are required for systems that are implemented using MPUs. Most of the current MPU systems realize memory, graphics, and
peripherals through chipsets, such as Graphics and Memory Controller Hub (GMCH), I/O Controller Hub (ICH), or System Controller Hub (SCH). Although it incurs higher cost, they have the advantage of versatility such that the designer can decide on the amount of memory and I/Os needed to fit the task of the target systems.

A microcontroller (MCU) has a CPU in addition with a fixed amount of memory and peripherals on a single chip. Therefore, external memory or peripherals are not common for systems built from MCUs. In general, when compared to MPUs, MCUs have limited processing power and lack of graphic capability. Systems built from MCUs often require some I/O operations to read signals and turn on/off certain bits. Many MCUs are focused on specific end applications and integrate specific I/O to suit for example ADC, DAC, PWM, etc.

![Figure 1. Microprocessor (MPU) System](image1)

![Figure 2. Microcontroller (MCU) System](image2)

**Microprocessor vs microcontroller**

Microprocessor (MPU) = CPU
Microcontroller (MCU) = CPU + Peripherals + Memory

Where:
Peripherals = I/O Ports, Clock, Timers, UARTs, ADC/DAC…..
Memory = EEPROM, SRAM, EPROM, Flash…..

The architectures of MCU and the Intel® Atom™ processor are different in terms of instruction sets, registers, and memory categories. Despite the architectural differences, the beauty of software development using high-level programming language (such as C-language) is that the source code is portable between hardware architectures. High-level language compilers handle instruction sets and
register differences, and generate the machine code for the target processor architecture.

For a long time, the C language has been the high-level language of choice for embedded systems development. Widely available open source software is available for x86 architectures and other architectures including MCUs, which makes it relatively easy to escape from the baggage of legacy product architectures when considering new product designs.

**Industrial Automation**

Industrial control systems are used in many application domains, such as power plant control, factory automation, etc. In general, all are characterized by distributed processing functions with critical timing requirements which present challenges to the designers of real time system components, such as logic controllers.

**Timing Requirements and Design Challenges**

A distributed processing control system consists of logic controllers, sensors, actuators, interconnecting field buses, etc. There are many concurrent tasks running on a single controller such as PLC, and each task has its stringent timing constraints. Beyond the individual controller, there are also end-to-end event-response requirements across all distributed components that need to be satisfied.

**Distributed Controller Level:**
Multiple concurrent tasks are executed on a single processor, whereby each of them has its own timing constraints.

**System Level:**
End-to-end event-response timing constraints across multiple distributed components over the interconnected buses, including signal or message propagation delay.

Delays of a process control system are due to: (1) interrupt latencies, (2) signal or message propagation delay over the network, (3) overheads incurred by asynchronous periodic tasks, and (4) overhead incurred by task scheduling within the real-time operating systems (RTOS). This characteristic poses a challenging problem to system designers, especially on real time designs. Therefore, fast and predictable event-response time becomes an important consideration by PLC vendors during hardware and software selection.
Electro-mechanical relays were deployed for control systems before electronic control solutions were introduced. Relay serves as switching, timing, and multiplying mechanisms for input devices. However, it does not allow programming.

The first programmable controller was developed by General Motors in 1968. It offered programming and troubleshooting flexibility to engineers and technicians. The programming methodology was based on the ladder diagrams. Figure 4 depicts the flow of a PLC controller.

The input collected by data acquisition (DAQ) devices such as sensors, will be converted to digital values and processed by the PLC. PLC will then provide signals to actuator to take appropriate actions based on the result of its processing. These functions may or may not take place on the same processor. But, the total latencies for each stage must be within the system requirements.

**PLC I/O Scanning Architecture**

PLC hardware generally consists of a CPU, direct analog or digital I/O, fieldbus interconnects, and serial communication interface.

Once booted up by initialization firmware, PLC enters infinite loops with a sequence of control instructions (referred to as "input scan") that can be configured by the programming system to which it is connected. Besides executing "input scan", PLC performs round-robin control and timer tasks, and then provides the results to the actuators. It is designed to run through the loops as fast as it can, and its response time is determined by the worst case loop period.

General-purpose operating systems (GPOS) such as Linux and Windows are typically architected to
maximize throughput which results in averaging of performance for a broad range of general purpose applications.

Designs that target general-purpose applications also decrease the predictability of the system's behavior and have significant impact on real-time performance.

Real-Time Operating Systems (RTOS), which are specifically architected to guarantee real-time performance and determinism have evolved as an alternative to general-purpose operating systems in many applications, including industrial control applications. Industrial control systems are typically implemented using PLCs running small RTOS such as VxWorks or QNX.

Figure 5. PLC I/O Scanning

Figure 6 shows that control and monitoring applications require an average interrupt response of 506.3µs. PLC used for motion controls or machine controls may have more aggressive interrupt latency requirements in the range of 150µs - 200µs.
Intel announced the low power Intel® Atom™ processor in March 2008. The micro-architecture is designed specifically for low power and small devices. It has a thermal design power (TDP) specification in the 0.6-2.5 watt range and can scale to 1.8-GHz speeds depending on customer need.

The Intel Atom processor was specifically designed to enable mobile internet devices and other devices requiring low power and small form factor. Consequently, the Intel Atom is increasingly being adopted for a wide range of embedded applications including industrial control.
Interrupt Architecture Analysis

Interrupts are events that indicate condition change in the system, processor, or current executing tasks. The condition change typically requires processor attention. Therefore, interrupts invoke a special software handler known as Interrupt Service Routine (ISR) which is programmed with specific actions taken by processor in response to the interrupt.

Interrupt triggering and response mechanisms are essential in a PLC application performing I/O scanning as described in the earlier section. It is a critical requirement that I/O scanning is completed within a consistent bounded time limit. It should be noted that the capability or product positioning of a PLC is partly determined by I/O scanning rates.

This section explains the interrupt architecture of platforms featuring the Intel® Atom™ processor, namely Intel® Z5xx Atom™ processor and Intel® US15W System Controller Hub (SCH) platform. Following is a summary of interrupt types supported by this platform:

**Peripheral Interrupt Request (PIRQ/INTR/INTx):** interrupt request source from peripherals, such as PCI devices, serial peripheral devices, etc.

**Serial Interrupt Request (SERIRQ):** interrupt request source from serial peripherals, such as keyboard, mouse, serial interface, etc.

**Non-Maskable Interrupt (NMI):** the highest level of severity which will cause system halt, such as memory corruption.

**System Management Interrupt (SMI):** typically used for system or platform level management without OS awareness, such as thermal sensor events, System Management RAM access, chassis open, etc.
System Control Interrupts (SCI): generally used by hardware to notify the OS about ACPI state events.

In general, interrupt events can be delivered from peripherals to CPU through either (1) legacy interrupts, or (2) Message Signaled Interrupts (MSI).

Legacy interrupt delivery method is implemented through a "side-band" INTR pin which is connected to the CPU through a pair of 8259 PIC controllers. In order to determine the ISR vector, the CPU has to acknowledge the 8259 PIC controller upon receiving INTR signal.

On the other hand, the MSI delivery method as specified by the PCI Special Interest Group (SIG) task force, is an "in-band" message write by a PCIe device. This method is realized by IOxAPIC or PCIe devices. Since it is "in-band" in nature, the ISR vector is carried as part of the message written by the peripheral, and acknowledgment cycles involved.

Interrupt Flow Analysis

This section provides an overall idea about different flow of interrupt delivery methods. Three cases of interrupt flows are described.
Figure 9. Intel® Atom™ Platform Interrupt Architecture

**Method 1: Legacy Interrupt Flow**

1. I/O device triggers an interrupt to 8259 PICs

2. 8259 PICs delivers "side-band" INTR interrupt signal to CPU
3. CPU sends interrupt acknowledge to 8259 PICs

4. 8259 PICs sends 8-bit interrupt vector to CPU

5. CPU pushes current states into stack and fetches ISR instructions from memory based on interrupt vector

**Method 2: IOxAPIC-MSI Interrupt Flow**

![Figure 11. IOxAPIC-MSI Interrupt Flow](image)

1. I/O device triggers an interrupt to IOxAPIC

2. IOxAPIC generates "in-band" MSI message and writes to CPU through backbone bus

3. CPU pushes current states onto the stack and fetches the ISR instructions from memory based on the interrupt vector

**Interrupt Flow Analysis (cont.)**

**Method 3: PCIe-MSI Interrupt Flow**
1. PCIe device generates "in-band" MSI message and writes to CPU through backbone bus

2. CPU pushes current states onto the stack and fetches the ISR instructions from memory based on the interrupt vector

**Interrupt Latency Comparison**
In the above three cases, it can be observed that Method 3 takes the simplest path. Furthermore, “in-band” MSI carries the interrupt vector in the MSI message packet and eliminates unnecessary cycles to fetch the interrupt vector. Therefore, for best real-time latency performance, MSI is the recommended choice of interrupt implementation.

Method 2 is similar and may be considered as an alternative to for legacy PCI devices that do not support MSI.

The following table depicts the comparison of interrupt latencies between IOxAPIC-MSI and PCIe-MSI.

Table 1. Interrupt Latency Comparison
<table>
<thead>
<tr>
<th></th>
<th>Worst Case (μs)</th>
<th>Average Case (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP_IO_APIC</td>
<td>5.85</td>
<td>4.18</td>
</tr>
<tr>
<td>SMP_IO_APIC</td>
<td>7.16</td>
<td>4.14</td>
</tr>
<tr>
<td>UP_MSI</td>
<td>3.60</td>
<td>1.66</td>
</tr>
<tr>
<td>SMP_MSI</td>
<td>3.55</td>
<td>1.58</td>
</tr>
</tbody>
</table>

Note: CentOS with kernel 2.6.18. Embedded Menlow platform with 1.6GHz CPU

UP_IO_APIC = single threaded, IOxAPIC-MSI

SMP_IO_APIC = multi threaded, IOxAPIC-MSI

UP_MSI = single threaded, PCIe-MSI

SMP_MSI = multi threaded, PCIe-MSI

**Local Interrupts (LINT0/LINT1) Usage**

The Local APIC that handles local interrupts, including Local Interrupt pins (LINT0 and LINT1), APIC timer, performance monitoring counters, thermal sensor, and internal APIC error detector. This section will focus on Local Interrupt pins LINT0 and LINT1.

Apart from interrupt delivery methods discussed in the preceding section, the Intel® Atom™ processor offers external devices direct interrupt pins into the CPU through Local APIC controller, namely LINT0 and LINT1.

The mapping structure of Local Interrupts is different than the 8259 PICs and IOxAPIC. Vectors for LINT0 and LINT1 can be statically fixed by programming the Local Vector Table (LVT) as shown in Figure 14. Due to static interrupt vector assignment, Local Interrupts do not require additional cycles for vector fetching which "side-band" signals are used. Thus, LINT0 or LINT1 pins are expected to deliver interrupts with lower and with more consistent latency. Moreover, it reduces backbone bus traffic and arbitration because the need for vector fetching is eliminated in this case.
Figure 13. Local APIC Functional Block

The following procedure describes how to program the LINT0 and LINT1 pins on a processor. In this example, LINT0 is programmed to be the ExtINT pin and LINT1 is programmed to be the NMI pin.

The following constants are defined:

LVT1 EQU 0FEE00350H

LVT2 EQU 0FEE00360H

LVT3 EQU 0FEE00370H

SVR EQU 0FEE000F0H

Use the following steps to program the LINT[1:0] pins:

1. Mask 8259 interrupts.
2. Enable APIC via SVR (spurious vector register) if not already enabled.

    MOV ESI, SVR ; address of SVR
    MOV EAX, [ESI]
    OR EAX, APIC_ENABLED ; set bit 8 to enable (0 on reset)
    MOV [ESI], EAX

3. Program LVT1 as an ExtINT which delivers the signal to the INTR signal of all processors cores listed in the destination as an interrupt that originated in an externally connected interrupt controller.

    MOV ESI, LVT1
    MOV EAX, [ESI]
    AND EAX, 0FFFE58FFH; mask off bits 8-10, 12, 14 and 16
    OR EAX, 700H; Bit 16=0 for not masked, Bit 15=0 for edge
                ; triggered, Bit 13=0 for high active input
                ; polarity, Bits 8-10 are 111b for ExtINT
    MOV [ESI], EAX; Write to LVT1

4. Program LVT2 as NMI, which delivers the signal on the NMI signal of all processor cores listed in the destination.

    MOV ESI, LVT2
    MOV EAX, [ESI]
    AND EAX, 0FFFE58FFH; mask off bits 8-10 and 15
OR EAX, 000000400H; Bit 16=0 for not masked, Bit 15=0 edge
; triggered, Bit 13=0 for high active input

; polarity, Bits 8-10 are 100b for NMI

MOV [ESI], EAX; Write to LVT2

; Unmask 8259 interrupts and allow NMI.


Conclusion

This white paper gave an overview of the real-time nature of PLC applications and analyzed the interrupt architecture on platforms featuring the Intel® Atom™ processor.

Multiple interrupt flows involving Legacy 8259 PICs and IOxAPIC were described and latency comparison was provided. This paper recommended Local Interrupts (LINT0 and LINT1) that are supposed to deliver interrupts from external devices to the CPU at faster and consistent latency. This information is critical for PLC developers who have to ensure that real time requirements can be fulfilled.

In summary, it has been shown that the capability of the Intel Atom processor extends far beyond the scope of mobile internet devices to include embedded controllers with hard real-time constraints such as PLCs and many other embedded device categories.

For more, visit the Intel Embedded Design Center to access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community.

Author bio:
Lee Eng Kean is a Technical Marketing Engineer with Embedded & Communications Group (ECG) at Intel
Related links:
Guidelines for migrating to the Intel Atom processor from another processor architecture | Part 2: Example solutions
Optimizing Embedded Designs for the Intel Atom Processor
Understanding Assembly Language for IA-32 and Intel 64 Architectures - Part 1: Basic overview and instruction set extensions | Part 2: Effectively using assembly language listings
Fundamentals of x86 Architecture